Lowering
A Static Optimization Technique for Transparent Functional Reactivity

Master's Project Report

Kimberley Burchett
with Greg Cooper and Shriram Krishnamurthi advising

This is a brief description of the work I did in 2006 for my master's degree. For a more detailed description, the reader is encouraged to read the published paper [1] instead.

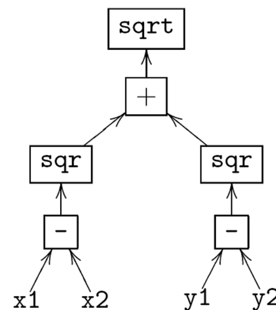
1. Background

Functional Reactive Programming (FRP) extends traditional functional programming by adding support for time-varying values, referred to as *behaviors*. For example, an FRP programming language might expose behaviors representing the current time, the current position of the mouse, the current temperature outside, etc. Note that the word "current" is common to all of these descriptions; the essential feature of behaviors is that their value is liable to change at any time.

FRP programs can work with behaviors just as if they were regular, constant values (e.g. they can be passed to functions, stored in data structures, etc), and whenever the value of a behavior changes, the relevant portion of the program will be *recomputed*. In short, a behavior is rather like a cell in a spreadsheet; whenever its value changes, any cell that refers to it will automatically be updated.

FrTime is a Scheme implementation of FRP, written by Greg Cooper. The main design goal for FrTime was that it should feel like Scheme as much as possible, so behaviors are implicitly supported throughout the entire language. The implementation of FrTime is based on the dynamic construction of an acyclic dataflow graph. Every node in the dataflow graph is a behavior. The leaves of the graph are the built-in behaviors provided by FrTime, and the interior nodes are behaviors that use other behaviors as inputs. Whenever a function is applied to a behavior argument, a new interior node is automatically constructed and added to the dataflow graph. For example, here is the FrTime code and the dataflow graph for the calculation of Euclidean distance.

```
(sqrt (+ (sqr (- x1 x2))
         (sqr (- y1 y2)))))
```



Because every single function application in a FrTime program can result in a new dataflow graph node, large FrTime programs can result in huge dataflow graphs, which means responding to changes can take an unacceptably long time. For example, a naïve port of the Slideshow program to FrTime resulted in slide decks which took several minutes to show the first slide, and which consumed hundreds of megabytes of heap space.

The goal of the project described here was to reduce the size of the dataflow graphs created by FrTime programs, in the hope that this would improve performance. I succeeded in this goal; for example, a benchmark of the TexPict library (upon which Slideshow is built) runs 78 times faster with optimization than without.

2. Lowering

FrTime supports behaviors throughout the entire language. It does this by replacing the primitive Scheme functions such as `+`, `-`, `*`, `/`, etc with versions that have been *lifted*. Lift is a higher-order function with the following type signature (where *t* and *u* represent arbitrary types):

$$\text{lift} : (t_1\ t_2\ \ldots\ t_n \rightarrow u) \rightarrow (\text{behavior}(t_1)\ \ \text{behavior}(t_2)\ \ldots\ \text{behavior}(t_n) \rightarrow \text{behavior}(u))$$

It is this implicit lifting of all functions that causes the construction of a new dataflow node for every function application. The lowering optimization is so named because it works by undoing this implicit lifting wherever possible. Specifically, the optimizer rewrites source code expressions so that instead of making several calls to individually-lifted functions, the program makes a single call to a lifted function which is composed of calls to several non-lifted (lowered) functions. For example, the optimized version of the Euclidean distance expression shown previously is as follows:

```
((lift (lambda (x1 x2 y1 y2)
         (sqrt (+ (sqr (- x1 x2))
                  (sqr (- y1 y2))))))
 x1 x2 y1 y2))
```

Here, underlined functions indicate the "lowered equivalent" of the lifted function with the same name. The lowered equivalent of a function is a function that performs the same operation, but on non-time-varying values. For lifted functions, this is just the function originally passed to lift. Not all functions have a lowered equivalent. For example, the `derivative` and `delay` functions are inherently defined only over behaviors. Additionally, it may not be statically decidable whether or not a function has a lowered equivalent, for example when calling a first-class closure.

As you can see in the above example, in order to optimize an expression we must extract the set of free variables and pass them as arguments to the new, lifted function. This is necessary because we can't always statically determine whether a free identifier will evaluate to a behavior or a constant, and we must ensure that no behaviors are passed to non-lifted functions.

The optimizer must keep track of the correspondence between lifted functions and their lower equivalents. This is done by maintaining a mapping from lifted identifiers to lowered equivalent identifiers. The mapping is initialized with entries for all the built-in FrTime functions. The mapping is extended for any user-defined functions whose lower equivalent can be statically constructed (i.e. any user-defined function whose entire definition can be optimized, and which does not refer to any global variables). The ability to extend the mapping for user-defined functions is critical for achieving good performance.

In order to make the lowering transformation incremental and compositional, I defined a new syntactic primitive called *dip*. Using dip, the optimized version of Euclidean distance looks like this:

```
(dip (x1 x2 y1 y2)
   (sqrt (+ (sqr (- x1 x2))
            (sqr (- y1 y2)))))
```

Unlike lifting or lowering, dipping does not change the observable behavior of a segment of code. This means that it can be done in a bottom-up fashion, coalescing dipped subexpressions into a single dipped expression. Even if an expression cannot be dipped (either because it is a call to a function with no known lowered equivalent, or because it is an unsupported syntactic form), its subexpressions may still be dippable, yielding some performance improvement.

For a formal definition of the optimization, see the published paper [1]. I designed the system of inference rules described there, and there is no need to duplicate it here.

3. Performance Results

I chose four benchmarks to evaluate the effectiveness of the optimizer. The results are shown below.

	Count	Needles	S'sheet	TexPict
Size (exprs)	7	62	2,663	13,022
Start_{orig} (sec)	9.5	89.0	9.2	35.2
Start_{opt} (sec)	<0.1	35.3	11.8	28.9
Mem_{orig} (MB)	204.7	581.4	34.8	170.7
Mem_{opt} (MB)	0.2	240.5	50.9	119.4
Shrinkage (ratio)	*971*	*2.4*	*0.7*	*1.4*
Run_{orig} (sec)	4.8	5.6	19.3	273.4
Run_{opt} (sec)	<0.1	2.0	20.5	3.5
Speedup (ratio)	*16,000*	*2.8*	*0.94*	*78.1*

Count is a microbenchmark designed to show the maximum potential of the optimization, for small recursive functions that can be completely lowered. The benchmark simply counts down from a number *n* to zero, and then back up to *n*.

Needles is an interactive program written by Robb Cutler prior to this work. The benchmark displays a 60x60 grid of short line segments that swivel to follow the mouse. Optimization nearly triples its performance, and the main remaining bottleneck is rendering time (not FrTime overhead).

Spreadsheet is a from-scratch implementation of a spreadsheet in FrTime. The optimizer had very little effect on performance, and it increased memory usage noticeably. In versions of DrScheme prior to 360, the optimizer had shown a modest improvement for this benchmark with optimization, so it's not clear why that benefit has disappeared.

TexPict is the image compositing library underlying the Slideshow program whose abysmal performance in FrTime provided the initial motivation for this project. As you can see, optimization provides dramatic performance improvements even on large, real-world programs written without FRP in mind. Even after optimization the reactive version is significantly slower than the non-reactive version, but the speedup of nearly two orders of magnitude enables the use of modest amounts of reactivity where it had been impractical before.

4. Discussion

The benchmark results show that lowering can provide significant benefits, even to real-world programs. Since the choice of whether or not to use optimization it is up to the FrTime developers, cases that show degraded performance (such as Spreadsheet) can simply avoid using the optimizer. Furthermore, since optimized modules are compatible with non-optimized modules, optimization can be limited to those modules where it's beneficial.

The idea behind lowering is relevant beyond just FRP. In general, the optimization described here should apply to any monad where lift distributes over function composition (that is, where lift f . lift g = lift (f . g)). This suggests the idea may be useful to languages that use monads extensively, such as Haskell.

At an abstract level, lowering is similar to deforestation in that it eliminates the need to use intermediate data structures to pass values between expressions. The implementation details are very different, however.

5. References

[1] K. Burchett, G. H. Cooper and S. Krishnamurthi. Lowering: A Static Optimization Technique for Transparent Functional Reactivity. *Partial Evaluation and Program Manipulation*, pages 71-80, 2007.